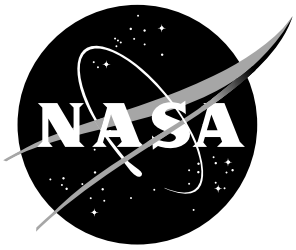
NASA/TM–20220005189



PCLAM: a Python Module for Computing Surface Lineloads and Moments

Michael W. Lee; T.J. Wignall
Langley Research Center, Hampton, Virginia

May 2023

NASA STI Program...in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI Program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI Program provides access to the NASA Aeronautics and Space Database and its public interface, the NASA Technical Report Server, thus providing one of the largest collection of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

For more information about the NASA STI Program, see the following:

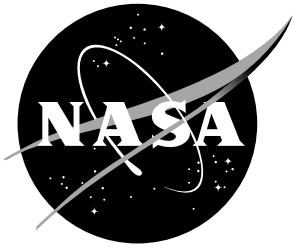
- Access the NASA STI program home page at `http://www.sti.nasa.gov`

- E-mail your question to `help@sti.nasa.gov`

- Phone the NASA STI Information Desk at 757-864-9658

- Write to:
  NASA STI Information Desk
  Mail Stop 148
  NASA Langley Research Center
  Hampton, VA 23681-2199

NASA/TM–20220005189



PCLAM: a Python Module for Computing Surface Lineloads and Moments

Michael W. Lee; T.J. Wignall
Langley Research Center, Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

May 2023

Available from:

NASA STI Program / Mail Stop 148
NASA Langley Research Center
Hampton, VA 23681-2199
Fax: 757-864-6500

Abstract

Lineloads serve a unique and important role in aerodynamic database development as well as configuration design and analysis. A new software suite was constructed that can compute lineloads rapidly enough that the calculations can run in tandem with high-fidelity fluid flow solvers. This enables the calculation of iteration- or time-dependent lineloads, which have thus far been too computationally costly to create for complicated systems like launch vehicles. The suite is organized into a standalone Python module named PCLAM (PCLAM Computes Lineloads And Moments) that can be imported into other software with minimal restructuring by the user or developer. The computed lineloads are integrated with a C^0 numerical quality and exhibit the expected sensitivity to underlying grid resolution. Even at low grid and lineload resolutions, the computed lineloads were found to be in strong agreement with several analytical test cases.

Nomenclature

A_n	area of subelement n
b_p	bin p as defined along the discretized lineload axis
C_P	pressure coefficient
C^0	integration approximation in which each discretized component is taken to be constant
f_m	information stored at a particular point m on an element
$\bar{f}_{m,n,p}$	arithmetic mean of information stored at points m, n, p on an element
$\overrightarrow{ll}_d$	continuous lineload vector as defined along the dimension d
$\overrightarrow{lm}_d$	continuous line moment vector as defined along the dimension d
$\hat{ll}_d$	discrete bin-normalized lineload vector as defined along the dimension d
$\hat{lm}_d$	discrete bin-normalized line moment vector as defined along the dimension d
L_{ref}	reference length
$\hat{n}$	unit-length surface normal
N	number of points on an element
$\vec{r}$	moment arm
S	arbitrary surface
v_{mk}	value of interest m within bin k
w_n	interpolation weight for element node n
W	sum of all interpolation weights for an element
x, y, z	spatial dimensions
Δb	discrete lineload bin width
Δx	discrete x-axis step size
$\vec{\vec{\sigma}}$	fluid stress tensor

1 Introduction

Lineloads, also known as sectional loads, serve a unique role in aerodynamic database generation for complicated configurations. Integrated forces and moments can fail to provide enough information for structural tolerance assessment or controls calibration, thus motivating a more spatially distributed measurement of the data. A lineload, as its name implies, is a force (or moment) calculated by integrating the surface stresses in only two of the body's three dimensions. The result is thus a force or moment function plotted against the remaining axis, as exhibited in Figure 1.

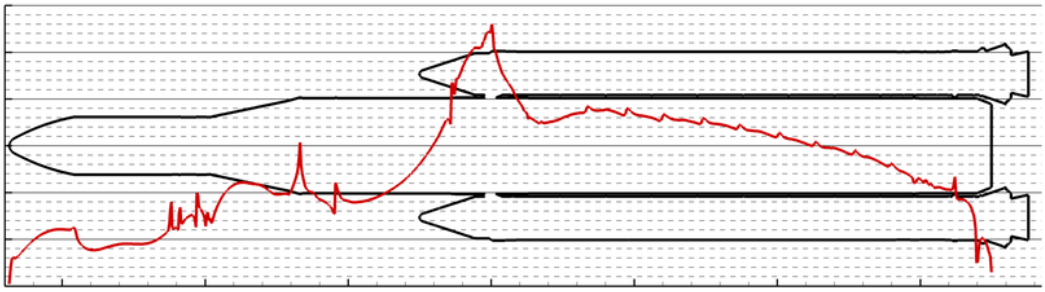


Figure 1: Normal force lineloads as computed on the centerbody of the Space Launch System during the transition stage of flight. Image taken from Ref. [1] with author's permission.

Typically, NASA teams have used TRILOAD, part of the Chimera Grid Tools suite, for their lineloads calculations [2]. TRILOAD is a versatile tool that handles cases where lineloads are needed such as the recent work of comparing lineloads derived from pressure-sensitive paint (PSP) data to computational fluid dynamics (CFD) Data [3]. However, there are enough limitations to TRILOAD that a new lineload computation tool was needed. The driving motivation was the lack of TRILOAD's ability to allow the user to directly separate pressure lineloads from skin friction lineloads (the two components of aerodynamic surface stress), each of which on its own can be used for different analyses. This was necessary for the efficient generation of a data-fused reduced-order model that was used for the development of a liftoff and transition lineload database for the Space Launch System (SLS) program [4]. As analysis of the liftoff and transition regime and advancement of the SLS program continued, the need for efficient calculation of time-dependent lineloads became apparent. The need for time-dependent lineloads was driven in part by interest in improved uncertainty quantification in lineloads databases, as well as the study of dynamic phenomena such as wind-induced oscillation. To complete this goal, it was necessary to develop a lineloads tool that can 1) be integrated directly with flow solvers and 2) operate as quickly as possible to not slow down the solver's iteration speed.

To the first point, the Kestrel flow solver [5], currently used for development of liftoff and transition lineloads databases for the SLS, has a python-based software development kit (SDK), which enables such a tool to be applied. The existence

of a python-based lineloads module would in this way enable per-iteration lineload calculations as part of the simulation itself, which would drastically reduce what data must be saved as part of the simulation process and in turn improve the computational cost of developing a more advanced lineloads database.

To the second point, in most cases, even while the flow solution varies in time and the volumetric discretization varies, the surface grids often remain the same. As such, an ideal lineloads tool would be able to separate the computational cost of preparing the surface mesh for lineloads integration from the actual integration itself. Such a framework would enable all lineloads calculated after the first iteration to be done so at a significantly reduced cost.

The resulting python suite performs discrete integration to yield lineloads along any one of the three principal axes of the provided geometry. As long as the provided surface mesh is triangular in nature, all grid adjustments are made internally as the lineload is computed. If the same grid is used again for a different flow solution, then the suite enables the user to skip the grid management in subsequent calculations and simply perform the integration immediately upon data ingestion. Doing so dramatically reduces the lineload calculation cost.

The remainder of this technical memorandum summarizes the theory and verification of the lineloads suite. A more detailed software user's manual is provided in Appendix A.

2 Lineload Definition and Integration

Lineloads are computed by integrating a stress tensor $\vec{\vec{\sigma}}$ defined on a surface of interest but retaining dependence in the lineload's defined axis. The integrated force on the surface would be this same integral without the retention of one surface axis. Analytically, a lineload in the x-direction would be computed on a surface S defined by normals $\hat{n}$ in (x, y, z) space by solving the integral

$$\overrightarrow{ll}_x \equiv \oint_S \vec{\vec{\sigma}}(x, y, z) \cdot \hat{n}\, dy\, dz \; . \tag{1}$$

The line moment is computed by the same integral with the inclusion of a moment arm $\vec{r}$:

$$\overrightarrow{lm}_x \equiv \oint_S \vec{r} \times \vec{\vec{\sigma}}(x, y, z) \cdot \hat{n}\, dy\, dz \; . \tag{2}$$

As the analytical surface stress tensor is rarely defined in applied problems, these integrals must be approximated through discretization. The lineload axis is broken into integration bins (see §2.1), within each of which the discrete surface stress tensor is integrated in all three geometric dimensions. The integrated values from all tensor components comprise the lineload. The lineloads are then normalized by bin width Δb (and scaled by reference length L_{ref}) to make them independent of the chosen resolution.

$$\hat{ll}_x \equiv \frac{\vec{ll}_x}{\Delta b / L_{ref}} \tag{3}$$

Figure 2 illustrates the employed method of integration within each bin. Generalized to apply in any Cartesian direction, it implements a mean subelement integration approximation. An element is subdivided by any bin boundaries that may apply and the fractional areas are applied to all relevant bins. Linear interpolation finds bin-bounded values, e.g., f_4 in the example element is computed based on f_1 and f_2. The subelement values applied to each bin are then the arithmetic means of the bin-bounded, possibly interpolated values, e.g., the subelement value in the blue (left) subelement in the figure is taken to be the arithmetic mean of f_1, f_4, and f_9. The element and subelement means, and their corresponding areas, can then be directly summed to yield an integrated value for each bin – accounting for the surface normal if necessary.[1]

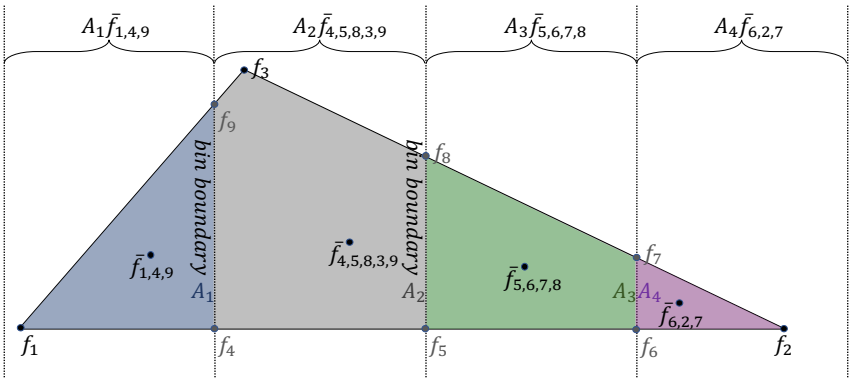


Figure 2: Treatment of node-centered element values during bin integration.

2.1 Bin Definitions

The lineloads are discretized along the axis of interest using one of two user-specified bin definitions. Figure 3 illustrates the difference in bin boundaries from these two definitions. Where one definition, labeled b_0 in the figure, discretizes the lineload axis such that bin centers correspond to uniformly spaced locations between the body's extrema, the other definition (b_1) does so such that the bins themselves span the body's extrema.

A benefit of the b_0 bins is that the geometry extrema are retained in the lineload regardless of bin number whereas the b_1 bins only ever asymptotically approach the geometry extrema. This comes at the cost, however, of storing "halved" values on each end by virtue of the half-empty terminal bins. This half-value can be adjusted by considering these edge-bins to be themselves virtually halved in width, but then an inconsistency exists in bin widths and that deviates from the database-standard midpoint bin reporting protocols. As defined in Equation 3, it is standard practice to use a uniform Δb when computing and normalizing the lineloads and thus an adjustment of the outer bin widths with the b_0 discretization is not recommended

[1] In standard application, the pressure is passed to the module as a scalar without surface normal information but the shear stress is passed as Cartesian components, which are assumed to correspond to the correct geometry.

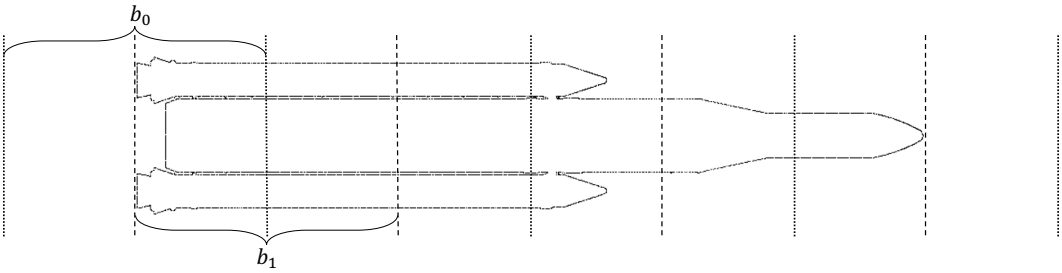


Figure 3: Bins and bin boundaries as defined by the old (b_0) and new (b_1) methods. Dotted and dashed lines illustrate the different bin boundaries.

for common usage.

The b_1 bins, by comparison, cannot report information exactly at the geometry extrema but also do not suffer from any idiosyncrasies in bin width or reported magnitudes. Mathematically, both the b_0 and b_1 bins are equally consistent with the C^0 integration approximation being performed between the geometry's extrema. However, with a constant bin width being used in the lineloads' normalization, the b_0 bins will always present "halved" values in the two outermost bins. This is observed in the test cases presented below, for example in Figure 7.

3 Interpolation of Node Values to Bin Boundaries after Initial Calculation

The linear interpolation of nodal surface data onto the bin boundaries is a computationally costly step – far more costly than the lineloads integration itself. As this interpolation is grid-specific and not solution-specific, a technique was developed to avoid it after the first grid-specific lineloads calculation.

Interpolation is a form of weighted averaging where the weights are determined by the distance from source points,

$$f(x_n) = \frac{1}{W} \sum_{i=1}^{N} w_i f(x_i) \ , \tag{4}$$

where $f(x)$ are the spatially distributed values of interest and w_i are the corresponding weights as the derived distance from x_i to x_n. W is simply the sum of all weights to normalize the result. In the case of the lineloads, interpolations are only ever performed linearly along element boundaries. Thus, only two (out of three possible nodes of a triangular element) weights must be found for each bin boundary point.

For a given x_n, the normalized weight of a given node, w_j, can be backed out by setting $f(x_j)$ to unity and setting $f(x)$ to be zero at all other locations.

$$f(x_n) = \frac{1}{W}(w_j f(x_j) + \sum_{i \neq j}^{N} w_i f(x_i)) \tag{5}$$

$$\frac{w_j}{W} = f(x_n) = \frac{1}{W}(w_j 1 + \sum_{i \neq j}^{N} w_i 0) \tag{6}$$

A simple augmentation of the surface data passed to the binning procedure on the first lineloads calculation accomplishes this as negligible cost increase using the existing mean-calculating procedure.

The lineloads integrator in this software suite does not formally partition elements into subelements, but rather tracks subcontributions in all relevant bins using the computed subelement means. For each element, a matrix of surface values v_{nm} can be created where m is the node index for the element and n is the value of interest, e.g., pressure versus the y-component of the skin friction. Performing matrix multiplication with the weight matrix constructed above, which itself has components w_{mk} for k integration bins, yields a matrix v'_{mk} of values of interest interpolated appropriately to the subelement centers of all relevant bins for that element.

$$\begin{bmatrix} v'_{11} & & v'_{1k} \\ v'_{21} & \cdots & v'_{2k} \\ v'_{31} & & v'_{3k} \end{bmatrix} = \begin{bmatrix} v_{11} & v_{12} & v_{13} \\ & \vdots & \\ v_{m1} & v_{m2} & v_{m3} \end{bmatrix} \begin{bmatrix} w_{11} & & w_{1k} \\ w_{21} & \cdots & w_{2k} \\ w_{31} & & w_{3k} \end{bmatrix} \tag{7}$$

These sparse matrices can then be used directly to perform the lineload integration. Calculation of the weighting matrix when the elements are first binned and interpolated thus allows the same grid to yield a lineload of the same resolution multiple times with different surface data without the need to perform the interpolation each time.

4 Representative Examples

To validate results, two test geometries were designed: a flat plate and a tangent-ogive cylinder.

4.1 Flat Plate

The flat plate was defined in the xy plane. The plate was defined between -1 and 1 in each of two axes. Two different grid spacings were used, presented here in MatLab notation.

$$x_{uniform} \equiv -1 : \Delta x : 1 \tag{8}$$

$$x_{cubic} \equiv \frac{4}{5} x_{uniform}^3 + \frac{1}{5} x_{uniform} \tag{9}$$

The same distributions were used to define the grid in the y-axis. The resulting grids are shown in Figure 4. The applied plates had 101 points in x and 21 points in y.

Two pressure coefficient functions were defined on the flat plates, thereby yielding four flat plate cases with unique grid-pressure combinations. Shear values are set to

zero on the plates. These functions are defined as follows.

$$C_P^{wavy} \equiv \cos(3\pi x)\left(\sin(3\pi y) + \frac{1}{5}\right) \tag{10}$$

$$C_P^{step} \equiv \begin{cases} 1 & x \leq 0 \;\&\; y \leq 0 \\ 0 & else \end{cases} \tag{11}$$

These functions can be integrated analytically such that the computed lineloads can be compared to what they are in the resolution limit. These analytical functions are also used to confirm the accuracy of the total integrated forces and moments.

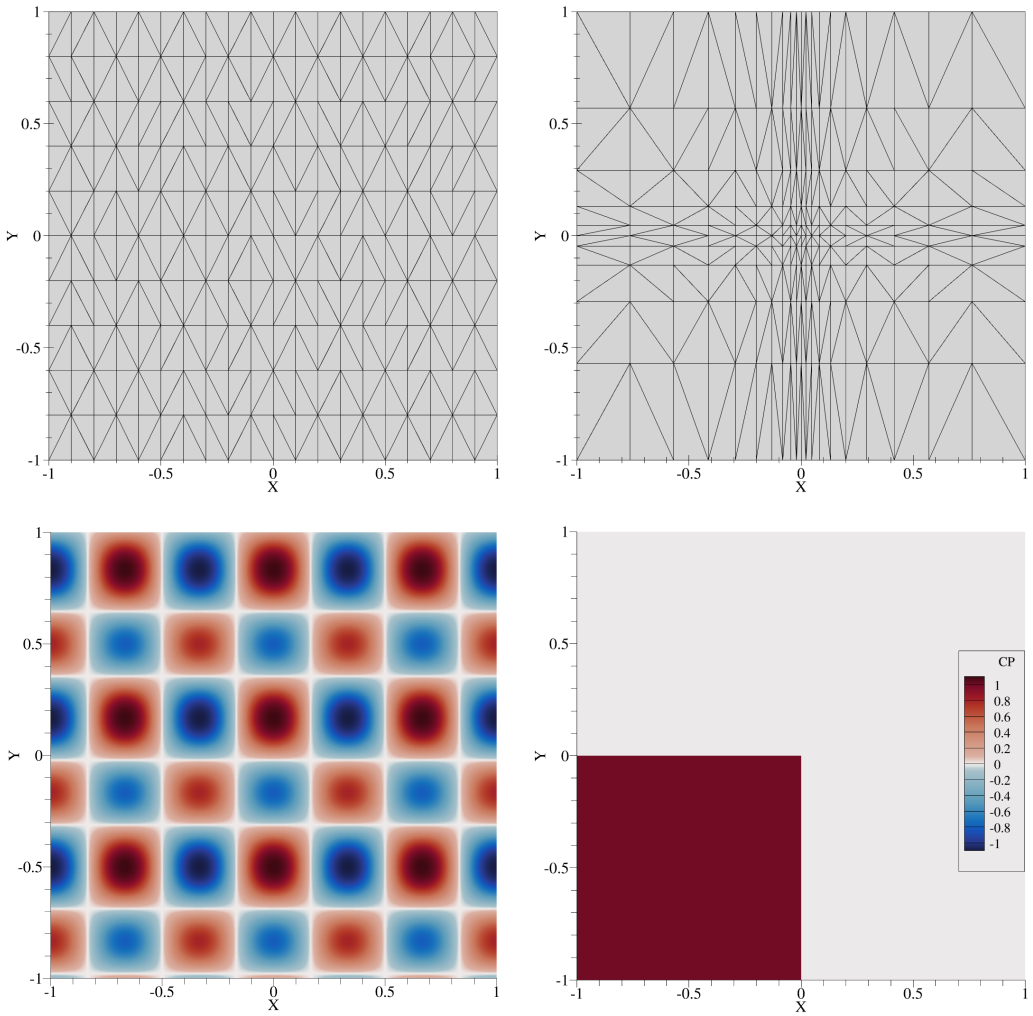


Figure 4: Sample flat plate grids (top): uniform (left) and cubic (right). Test pressure field functions (bottom). Resolution set to 21×11 points for visibility.

Figures 5, 6, 7, and 8 present lineloads for the four grid-pressure cases at two lineload resolutions. The halved edge values in the b_0 lineloads and the half-bin offset of the b_1 lineloads are apparent in all plots.

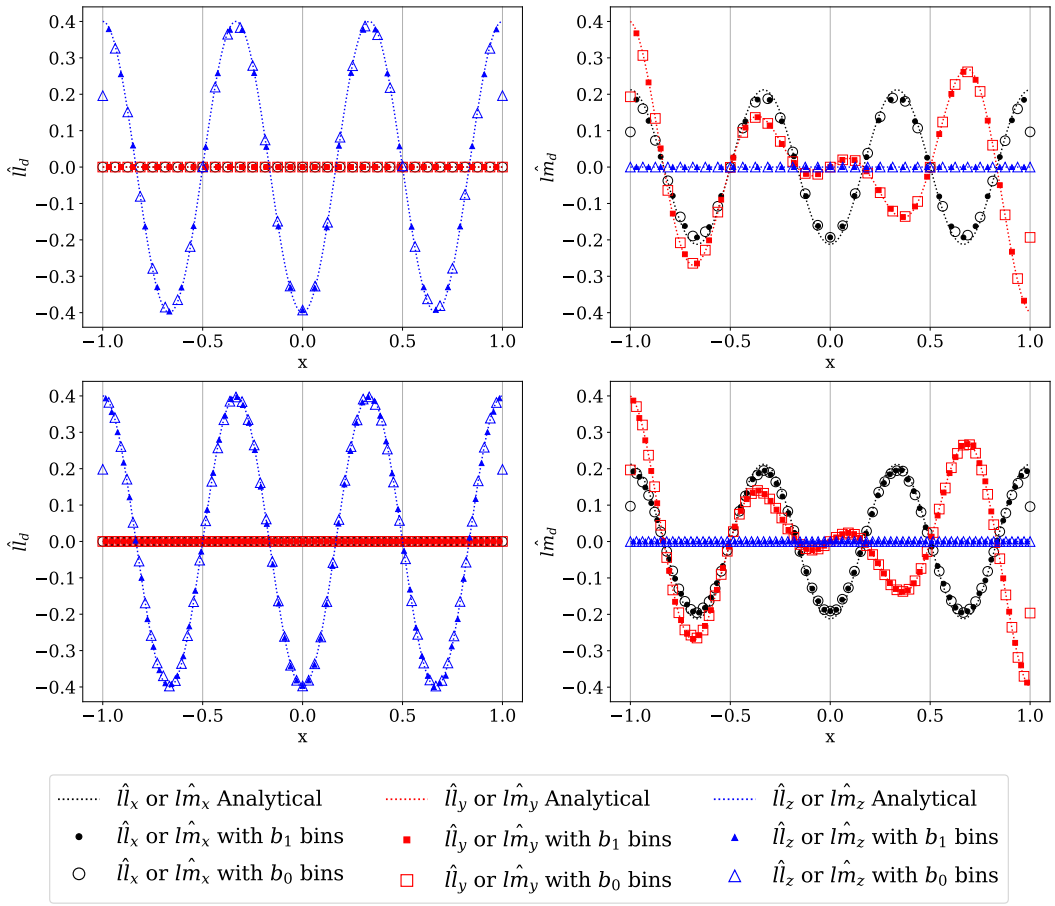


Figure 5: Lineloads (left) and line moments (right) computed on the uniform grid with the wavy pressure function. Lineloads have 33 (top) and 67 bins (bottom).

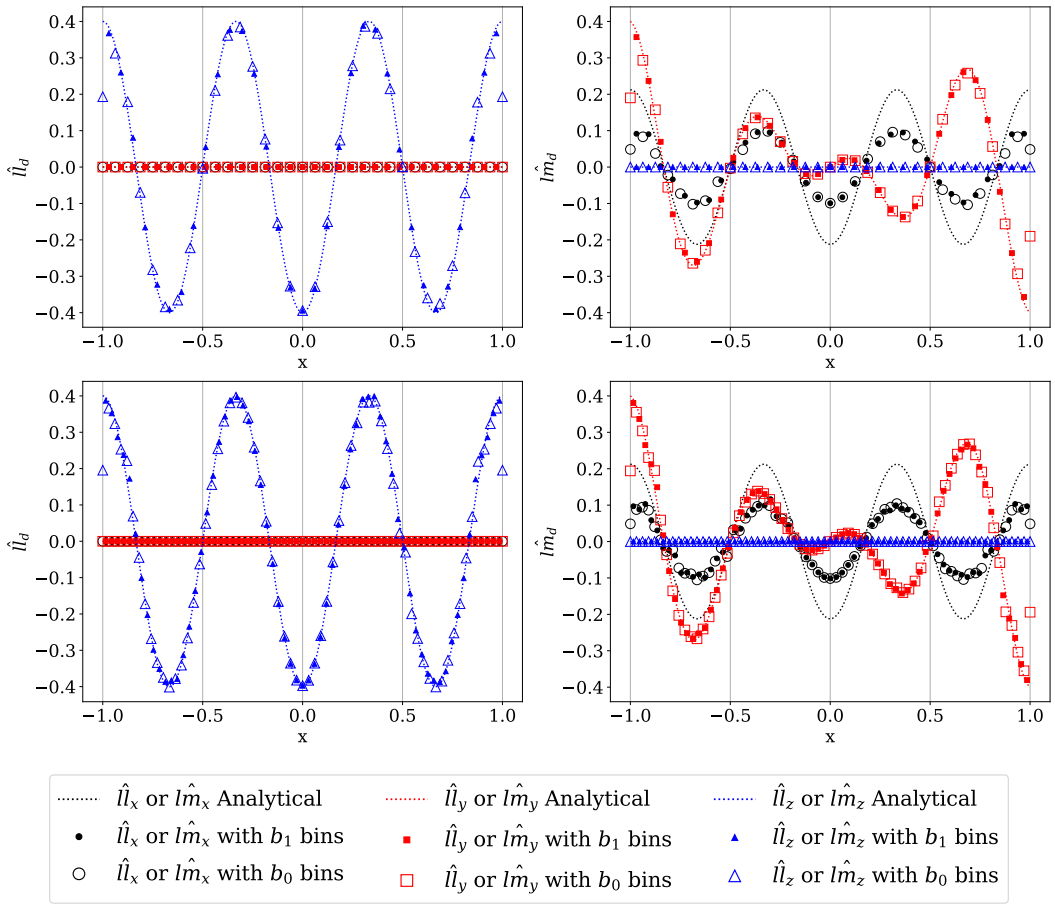


Figure 6: Lineloads (left) and line moments (right) computed on the cubic grid with the wavy pressure function. Lineloads have 33 (top) and 67 bins (bottom).

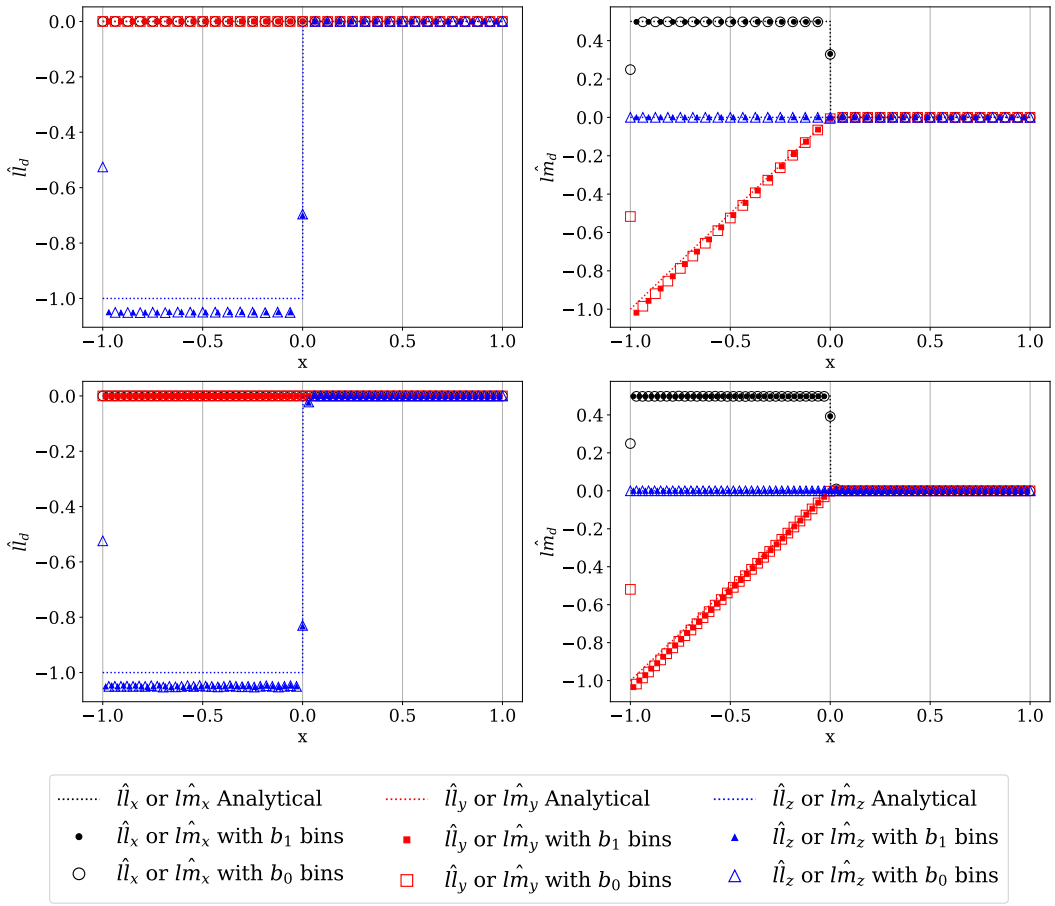


Figure 7: Lineloads (left) and line moments (right) computed on the uniform grid with the step pressure function. Lineloads have 33 (top) and 67 bins (bottom).

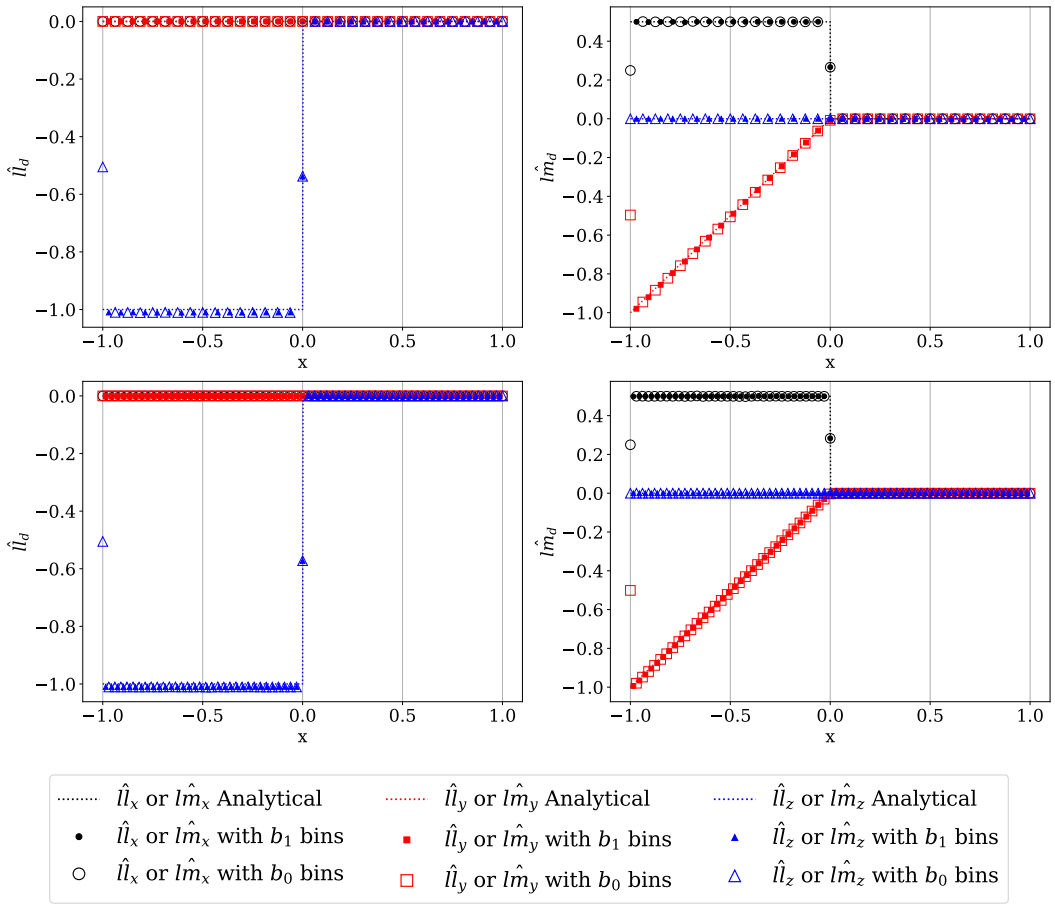


Figure 8: Lineloads (left) and line moments (right) computed on the cubic grid with the step pressure function. Lineloads have 33 (top) and 67 bins (bottom).

The consistent difference between analytical and computed "wavy" lineloads on the cubic grid, and likewise the "step" lineloads on the uniform grid, are a function of the limited resolution of the underlying grid. At the 101×21 point resolution selected for all of these cases, the sinusoidal pressure peaks were not consistently captured in the y-axis. As such, even with linear interpolation reducing the sensitivity to the number of lineload bins desired, the lineload's accuracy remains fundamentally bounded by the fidelity of the underlying grid data.

The flat plate was rotated into the xz and yz planes to ensure the code handled surfaces defined in any plane. Results are identical to those already plotted, except for those functions which are defined in the axes as the plate rotates.

4.2 Ogive Cylinder

An ogive cylinder geometry (shown in Figure 9) was defined to verify that the pressure and shear functions were handled separately and correctly. The pressure and shear components were specified as

$$\sigma_n \equiv (n+1)\cos((n+1)\pi x/10)\sin((n+1)\pi y/10) \tag{12}$$

where $n = 0$ corresponds to pressure and $n = 1,\ 2,\ 3$ corresponds to the x, y, and z shear components, respectively. This shear definition is understood to not remain tangent to the ogive surface at all times, meaning it is not a true shear stress. However, as the code handles surface normals when integrating the pressure but handles the three shear components as they are loaded in, the fact that the three-component shear stress is not formally shear does not invalidate the stress definition for verification purposes. In other words, this code like others is beholden to the fidelity of its input data.

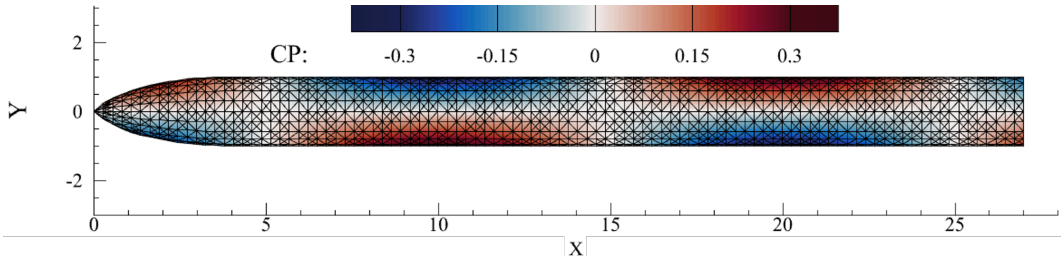


Figure 9: Tangent ogive cylinder geometry with surface mesh in black. Contours are of the artificial pressure coefficient defined in Equation 12.

More real-world verification of lineload axis specifications was also accomplished with this geometry as its 3D shape naturally exhibited lineloads in multiple axes at once. Figures 10 and 11 present a selection of pressure and shear lineloads in two axes of interest.

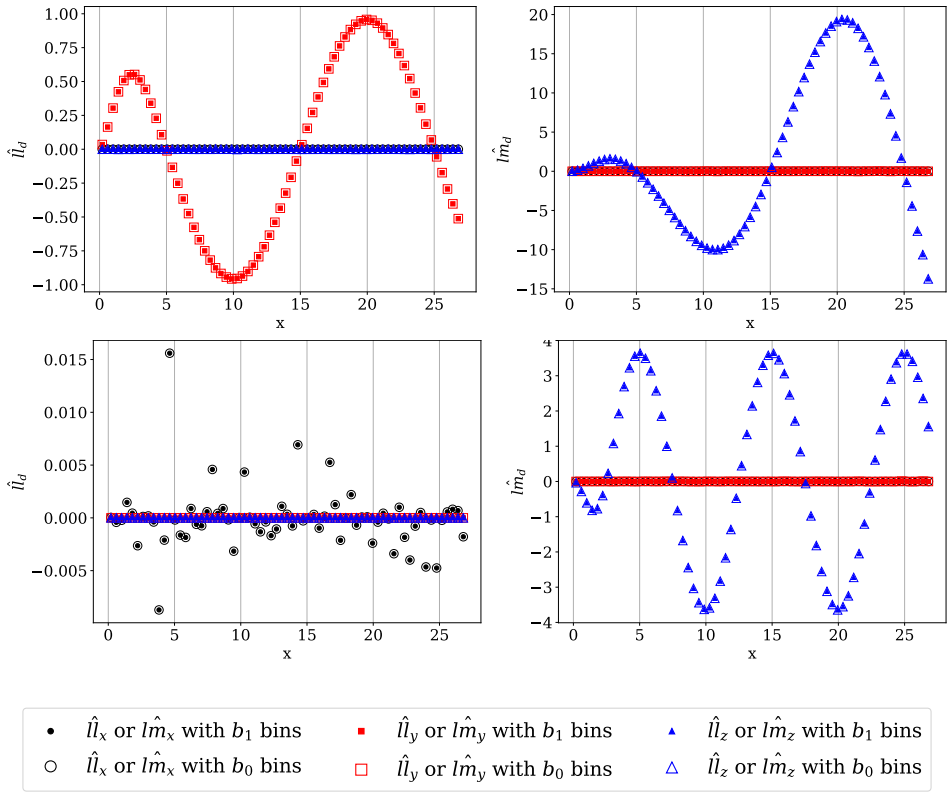


Figure 10: Pressure (top) and shear x-component (bottom) lineloads (left) and line moments (right) computed on the ogive geometry along the x-axis. All lineloads have 67 bins.

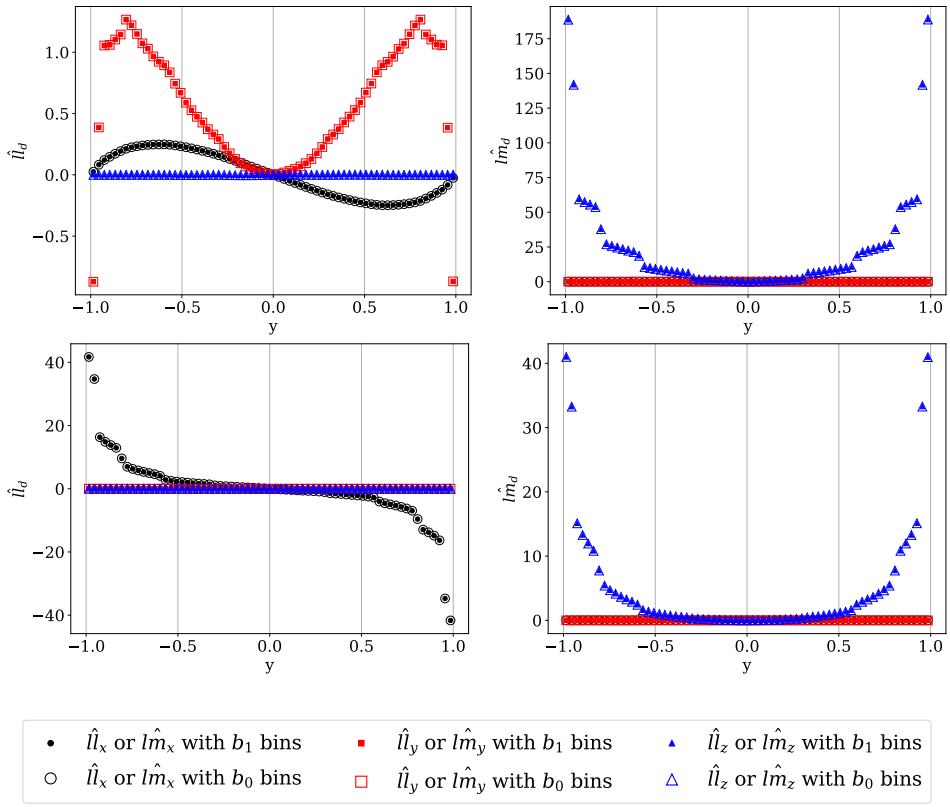


Figure 11: Pressure (top) and shear x-component (bottom) lineloads (left) and line moments (right) computed on the ogive geometry along the y-axis. All lineloads have 67 bins.

The noise in the shear x-lineloads (the bottom left of Figure 10) is another example of limitations in lineload quality caused by underlying low-resolution surface data. The discretized cylinder is not perfectly round, and thus when the analytically defined surface stresses are integrated over the relatively coarse mesh the resulting lineload is not numerically integrated to the exact zero. Increasing the resolution of the underlying surface mesh was found to reduce the amplitude of this error.

The ogive lineloads also illustrate the linear subelement interpolation performed as part of the procedure. The lineloads in Figure 11 were discretized to have 67 points along the y axis, but the cylinder's surface mesh had fewer than this many elements along that direction. As such, as part of the integration procedure, the elements were subdivided and their information was applied to multiple bins as described in §2. This is most clearly seen in the ogive y-line moments where the values follow linear trends before jumping as new elements are introduced to the next bins. As with the noise in Figure 10, the linear blocks in these line moments smoothed out when the surface mesh was refined.

5 Computational Cost

The computational cost of a single execution of the lineloads code naturally increases with surface mesh size and, to a lesser degree, lineload resolution. While a majority of the code's operations are vectorized, the bin decomposition and element subdivision operations are managed with nested `for` loops. These loops cause the code to run more slowly than would a fully vectorized framework as the loops are managed at a higher level of the interpreter than are the vectorized operations. However, a just-in-time (JIT) compiler available through a Python module called `numba` presents a way to speed up single-core execution in precisely this use case. The JIT implementation has little effect on the vectorized parts of the code but speeds up the parts that rely on nested loops. The relative speed of this lineloads suite is thus best characterized relative to whether the JIT compiler is enabled.

A flat plate made of 1,000,000 nodes (1,996,002 elements) was selected as a reference case for execution timings. The lineload was discretized using 200 bins. The time taken to create the same lineload, on a single core of a 2019 MacBook Pro laptop with a 2.4 giga-Hertz Intel Core i9 processor, was measured to be as presented in Table 1.

As expected, the lack of JIT-optimized function calls in subsequent same-grid calculations yielded no speed improvement from the use of `numba`.

Table 1: Time taken to compute the same lineload with various instances of the module. The just-in-time (JIT) compiler is available through the `numba` module.

	Without JIT (sec)	**With JIT (sec)**
Initial Calculation	550	115
Subsequent Calculations	70	68

While these calculation times are minimal compared to the CPU hours that it takes to simulate the surface stresses, when a time-dependent lineload is sought during each iteration of a larger simulation a single-core execution on the order of a minute is problematic. If the surface mesh is constant throughout the entire simulation, then the time to compute a particular lineload can be drastically reduced by performing the grid interpolation only once - on the first iteration - and then using the same weighting matrix on subsequent iterations to skip the binning and interpolating altogether.

6 Conclusion

Generation of increasingly detailed aerodynamic databases for configurations like the Space Launch System motivate the calculation of numerous lineloads during a single flow simulation. These iteration- or time-dependent lineloads will provide far more data than do the current time-averaged lineloads, thereby increasing both database confidence and fluid flow insight. However, calculating these time-dependent lineloads with present workflows would require an intractable amount of time due to 1) the cost of saving the required information from the flow solver and 2) the cost to compute each iteration's lineload.

The Python module, PCLAM, developed and summarized in this technical memorandum, addresses both of these limitations in existing lineloads workflows. Its organization as a standalone Python module enables it to be integrated directly with flow solvers like Kestrel through the flow solver's SDK. This integration yields a significant reduction in the lineloads' data read-write overhead. The reduction in computational cost on subsequent flow snapshots on a constant surface mesh reduces the lineloads integration time enough that it will not adversely affect the flow solver's runtime. Thus, it is possible for a time-dependent lineload for a particular fluid flow to be available virtually immediately when the fluid simulation is concluded.

References

1. B. W. Pomeroy and S. Krist. Verification and Validation of Kestrel DDES Simulations for SLS Transition Analysis. In *AIAA SciTech Forum, San Diego, CA*, AIAA 2022-1335, January 2022.

2. Shishir Pandya and William Chan. Computation of Sectional Loads from Surface Triangulation and Flow Data. In *20th AIAA Computational Fluid Dynamics Conference*, AIAA 2011-3680, June 2011.

3. Jamie Meeroff, Henry C Lee, Derek J Dalle, Stuart E Rogers, Nettie Roozeboom, and Jennifer Baerny. Comparison of SLS Sectional Loads from Pressure-Sensitive Paint and CFD. In *AIAA Scitech 2019 Forum*, AIAA 2019-2127, January 2019.

4. Thomas J Wignall. Liftoff and Transition Database Generation for Launch Ve-

hicles Using Data-Fusion-Based Modeling. In *AIAA Aviation 2019 Forum*, page 3401, 2019.

5. Scott A Morton, Brett Tillman, David R McDaniel, David R Sears, and Todd R Tuckey. Kestrel–A Fixed Wing Virtual Aircraft Product of the CREATE Program. In *2009 DoD High Performance Computing Modernization Program Users Group Conference*, pages 148–152. IEEE, 2009.

Appendix A

Software Organization

The PCLAM module is organized into a collection of functions saved within four objects:

1. `pclam.io.[function]`;
2. `pclam.util.[function]`;
3. `pclam.calc.[function]`;
4. and functions at the root level `pclam.[function]`.

The last of these contains the main operational functions of the suite. The other three objects contain, respectively, input-output, utility, and calculation functions. All functions that can be made faster with the `numba` just-in-time compiler are in `pclam.calc`. These functions perform a majority of the math and low-level data manipulation that comprises the suite. All functions that support general operation but that are not `numba`-compatible are in `pclam.util`. All functions that relate to loading or saving data in different formats at different points in the process are in `pclam.io`. These include loading TecPlot surface mesh files and saving ASCII lineloads data files.

In all, the PCLAM module requires the other following standard libraries to be available within the Python namespace: `datetime`, `json`, `numpy`, `os`, `scipy`, and `sys`. The `numba` module is optional. The `tecplot` module and associated user licenses are necessary only if the user attempts to handle binary or multi-zone ASCII TecPlot files, either for data input or data output.

The software is also distributed with unit tests which should verify if the package will operate as expected. Within the 'testing' directory, two scripts can be run to verify proper operation of all software components. These should only be run once PCLAM has been installed as a Python package on the user's machine. The script `test_unit.py` is run by calling `pytest test_unit.py`; it tests each PCLAM function individually but otherwise provides no output. The `plate_and_rocket_tests.py` file, again run with a `pytest` call, creates sample geometry files and generates and saves lineloads which can be visualized by the user. Any errors which occur when running these tests should be shared in detail by the `pytest` output itself.

A.1 Input Information

The `io` file also contains a `Config` class that tracks user settings and run options. For ease of access, the class is also loaded at the module level: `pclam.Config`. If the user does not specify any of the inputs, this class assigns default values. User options can be specified via an input file in the `.json` format. A brief description of all user inputs is provided below.

- `nll_points` defines the number of lineload points (default 100);
- `use_bin_mapping_file` determines whether the bin mapping matrix and bin areas will be saved and/or loaded for faster re-calculation on an existing surface grid (default False);
- `axis` defines the xyz-ordered axis along which the lineload will be defined (default x);
- `profile_axis` defines the xyz-ordered axis on which a reference profile is generated from the surface, for plotting reference (default z);
- `bin_edges_on_minmax` determines which bin method is to be used: bin edges on the body's extrema or bin centers on the body's extrema (default True, i.e., edges on extrema);
- `output_dir` sets the directory to which the lineloads will be output (default is the operating directory);
- `mapping_file_dir` sets the directory to/from which the bin mapping matrix and bin areas will be saved/loaded (default is the operating directory);
- `base_name` is the base file name to which other run details will be appended (default "placeholder");
- `mapping_file_name` is the file name header for the weighting matrix and bin areas saving/loading procedure (default "placeholder");
- `mrp` denotes the moment reference point in xyz as three decimals (default '[0,0,0]');
- `Lref` denotes the reference length with which the lineloads are nondimensionalized (default 1);
- `Sref` likewise denotes the reference area (default 1);
- `print_fandm` informs the code whether the user wants the integrated forces and moments to be saved as well as the lineloads (default True);
- `output_type` denotes the format for the lineload files (default ".dat", which is ASCII and TecPlot-compatible);
- `variables_saved` lists a subset of `[all, cp, cfx, cfy, cfz]` and denotes which lineload components are to be saved, with `cf_` denoting the shear components (default "all");
- `variables_to_load` labels the variables, in order, which are to be loaded (should be the less than or equal to the number of variables in the surface stress file) with labels consistent with what labels exist in the input file if it itself is labeled, as is the case with TecPlot (default `[x, y, z, cp]`);
- `absolute_pressure_integration` denotes whether the pressure coefficient data should be integrated in the tared form (standard) or with a correction for the absolute pressure, as is necessary with unclosed surface geometries (default False);
- `pinf_by_qref` denotes the reference absolute pressure with which the loaded pressure coefficient data must be corrected if specified by the user (default 0).

A.2 Running the Module On Its Own

In the standard application, a user can compute the lineloads on a given set of surface data by calling `pclam.run_pclam` and passing the necessary arguments. A sample

implementation is included with the software release in the form of an additional Python script: `run_pclam_sample.py`. This script takes arguments including a job name, an input file (a sample of which can be auto-generated), and a surface data file name. If the data file is in the `.npy` format, it is assumed that the surface data will be organized in a node-centered format and that an associated element connectivity file will share the same filename with `_data.npy` replaced with `_conn.npy`.